Journal of Multidisciplinary Engineering
Published by DigitoPub | Digitodontics International Academy
DigitoPub
Open Access & Peer-reviewed
Indexed in: Crossref, ICI, Google Scholar

Performance Comparison of Dapper and Entity Framework Core: Execution Speed, Memory Consumption, and CPU Usage

Amal Elbuaishi[1]* | Juma Ibrahim[1] | Ayad A. Keshlaf[2]

[1]Postgraduate office, College of Computer Technology, Tripoli, Libya.
[2]Computer Engineering and Information Technology Department, Faculty of Engineering, Sabratha University, Sabratha, Libya.

ABSTRACT

Purpose: This study aims to compare Entity Framework Core (EF Core) and Dapper in terms of performance, usability, and features to determine the most suitable option based on project requirements.
Materials and Methods: A test application was developed using a relational database containing one million records, including all modeled relationships. Complex CRUD (Create, Read, Update, Delete) operations were executed under controlled conditions. Key performance metrics such as execution time, memory usage, and CPU efficiency were measured and analyzed.
Results: The findings indicate that Dapper outperforms EF Core in execution speed while consuming fewer memory and CPU resources, making it more suitable for performance-critical applications.
Conclusion: Dapper is the preferred choice for applications where performance is a priority, whereas EF Core is more suitable for scenarios emphasizing developer productivity and maintainability. These results provide guidance for selecting the appropriate framework based on specific project needs.

Keywords: Object-Relational Mapping (ORM), Entity Framework Core, Dapper, Performance.

Audioabstract:

ARTICLE INFO

*Corresponding author:
Amal Elbuaishi
Email: ab2403006@cctt.edu.ly

Received: 12/02/2026
Accepted: 20/03/2026
Published: 24/03/2026

DOI:
https://doi.org/10.26629/jme.2026.05

Copyright: © 2026 by the authors.
Articles published in the *J Multidiscip Eng* are licensed under a Creative Commons Attribution-Noncommercial 4.0 International License.
https://creativecommons.org/licenses/by-nc/4.0/

How to cite this article | Elbuaishi A, Ibrahim J, Keshlaf AA. Performance comparison of dapper and entity framework core: Execution speed, memory consumption, and CPU usage. J Multidiscip Eng. 2026;1:31-38.

1. INTRODUCTION

Engineering In modern software development and the increasing volume of data, effective interaction with databases is critical to ensuring scalability and application performance.[1] Object-Relational Mapping (ORM) frameworks have emerged as essential tools that facilitate communication between relational databases and object-oriented program-ming languages by abstracting SQL queries into high-level operations.[2,3]

ORM frameworks vary in detail, but typically store an data model for generating SQL from the data model.[3] ORM solutions abstract the complexity of SQL operations and database interactions, allowing developers to focus on application logic while maintaining data consistency, portability, and productivity.[5] Among the most widely used ORM frameworks for .NET development are Entity Framework Core (EF Core)

and Dapper. EF Core is a full-featured ORM that offers automated change tracking, LINQ queries, and migration support, making it a powerful tool for developers who prioritize maintainability and abstraction.[3,6] Dapper, on the other hand, is a simple ORM designed to access high-performance databases, providing straightforward SQL execution with minimal overhead.[2] The choice between these frameworks significantly impacts application efficiency, especially in scenarios involving large datasets and complex relationships.[7,8] Despite the extensive use of ORM frameworks, performance concerns remain a critical challenge. Database performance refers to the speed and efficiency of retrieving, inserting, updating, and deleting data from a database.[7,9] Many factors affect performance, including query complexity, data size, and relationship structures within the database schema [9]. The results provide valuable insights to software engineers, developers, and database administrators, enabling them to make informed decisions when selecting the most appropriate ORM framework based on performance and scalability requirements.[9,10] The application lifecycle and code complexity are also considered for long-term system development.[11] This ensures database systems stay efficient even as the application expands.[10]

The rest of this paper is organized as follows: Section 2 introduces the relevant literature. Section 3 describes the methodology. Section 4 presents the obtained results. While section 5 discusses the research findings. Finally, Section 6 concludes the findings of this work.

Related Work

Wiatrowski, T. in[1] compared three popular ORM systems used in .NET: EF Core, Dapper, and DB to LINQ in terms of performance and memory consumption. A simple database was designed consisting of tables without relationships, tables with 1:1 relationships, tables with 1:N relationships, and tables with M:N relationships. The BenchmarkDotNet framework was used, which repeats each operation multiple times and calculates the average execution time and average memory usage.[12] The study[1] presented its results in the form of graphs of the average execution time of operations and average memory consumption according to the type of operation (delete, modify, add, and view) and according to the type of table relationship for data containing one or more records. All systems showed strengths and weaknesses according to the type of database operations and the type of table relationship. EF Core showed particularly good performance in operations that involve inserting multiple records at once. However, for other operations, it showed lower performance compared to other systems. The study proved that Dapper is more efficient in most types of operations. It performed particularly well in retrieving and deleting records. DB to LINQ showed stable performance in all types of operations and was not good in most tests. In terms of memory consumption during database operations, Dapper was the least consuming in most cases, while EF Core had the highest consumption, especially for operations that required processing a large number of records. DB to LINQ was particularly efficient in memory usage during multi-record fetch operations. In other cases, memory allocation was larger than Dapper's, and at an acceptable level.[1]

Y. Syamand in,[1] the performance of EF Core 6 and Dapper in performing CRUD operations in the .NET environment was compared in terms of execution speed and memory consumption. Performance Counter, BenchmarkDotNet and JMeter tools were used to measure each operation (add, read, update, delete) in each tool. The results indicated that Dapper outperforms read queries, with faster response times compared to EF Core 6, especially when dealing with big data queries. Despite the speed advantage, the study showed that EF Core 6 was more efficient in terms of memory consumption in some operations, such as POST requests. In contrast, Dapper showed slightly higher memory consumption in some operations due to the direct management of the connection to the database. For PUT and DELETE operations, Dapper outperformed in terms of speed, but memory consumption was slightly higher compared to EF Core 6. In terms of maximum response time, Dapper was sometimes slower than EF Core 6 in some cases requiring complex

operations. The paper suggests that the decision to use one framework over another mostly depends on the type of process most frequently used in a system.[2]

M. Forsberg in,[6] evaluated EF Core and dapper against three metrics: execution time, memory allocation, and code metrics. The tested queries were eight, four of which were basic CRUD operations and four were complex transactional queries. Each query was run 100 times to obtain average metrics, Benchmark.NET was used for performance metrics and signal testing to check statistical significance, and CodeAnalysis. Metrics was used for code metrics. The study showed that dapper generally outperforms on all metrics. However, the EF Core, with its available features, provides better implementation phase along with documentation and outperforms dapper in other areas related to ease of development such as entity relationship management, asynchronous query execution, automatic migrations.[9] Which means that if the primary goal of the application is targeting the performance, dapper is the better choice, but if development efficiency is important, EF Core is a suitable alternative.[6]

Zmaranda, D., et al in[7] compared three popular ORM systems used in .NET: EF Core, Dapper, and nHibernate in terms of performance and memory consumption. A database was created that included all types of table relationships (one-to-one, one-to-many, and many-to-many), to test and monitor the impact of the performance of these relationship types on different operations. Calibration was performed using the specialized library Benchmark DotNet. The number of entries used for testing was as follows: 500, 1000, 2000, 5000, and 10000 for each operation. Scenarios were set for each of (add - modify - delete - view) for all types of relationships (one-to-one, one-to-many) only to measure performance and memory consumption with the number of records mentioned for each tool.

The study showed that for add operations, nHibernate achieves the best results, followed by EF Core and then Dapper. For update operations, EF Core achieves the best results in one-to-one relationships; But in one-to-many relationships, Dapper achieves the best results. As for delete operations, when looking at one-to-one relation-ships, EF Core has the best execution time results followed by Dapper and then nHibernate; but in one-to-many relationships, nHibernate is better for a large number of entries (more than 1000) followed by EF Core and then Dapper. As for Get, the execution times of nHibernate and Framework Core are very close; for one-to-one relation-ships, Dapper gets better results as the number of entries increases, followed by EF Core and then nHibernate but in one-to-many relationships, nHibernate is better followed by Dapper, where EF Core performs worse as the number of entries increases. As for memory consumption, most results indicate that Dapper is better, followed by EF Core and then nHibernate.

P. Karabowicz, et al in[8] performed a comparative analysis of the ORM tools that are available in NuGet Manager, including Dapper, nHibernate, EF core, and OrmLite. The comparison of the tools was conducted on two relational database management systems, PostgreSQL and MySQL. Each tool performed eight Data Manipulation Language (DML) operations, including SELECT, INSERT, and UPDATE operations. The operations were repeated 1, 10, 100, 1000, and 10,000 times, respectively, and the largest number of scenarios were provided for query operations, where queries of varying levels of complexity were included. The execution time of each query was measured using the StopWatch class which is provided in .NET environment.

The results showed that the best ORM tool is Dapper. For most SELECT and UPDATE operations, Dapper turned out to be the fastest tool among the other tools. In contrast, for INSERT queries, NHibernate turned out to be the fastest ORM tool. However, for a larger number of iterations, Dapper executed the queries faster. Dapper, NHibernate, and OrmLite all have similar query execution speeds, regardless of the relational database management system used and the level of query complexity. For all query types tested, EF Core had the longest execution time, regardless of the number of iterations. This clearly indicates that this is the slowest ORM tool among those tested. A

review of previous studies identifies key research gaps that this study aims to address. Table 1 summarizes these gaps.

Table 1. Research gaps sumary.

Research Gap	Description	References
Impact of Big Data	The impact of big data volume on performance has not been analyzed	1,7,8
Processor Consumption Analysis	CPU consumption analysis has not been included	1,2,6
Complex Relationships:	The impact of data complexity on performance has not been tested	2,6,8
Query Complexity	Testing is limited to complex queries or standard real-world scenarios	6,7

2. MATERIALS AND METHODS

In order to be able to compare EF Core and Dapper on criteria such as execution speed and CPU consumption, the systems were run on different types of queries (Insert, Get, Update, and Delete) of varying complexity.

Development Environment

Define The development Environment tools as follows:

Visual Studio 2022.

Entity Framework Core 9.0.2.

Dapper 2.1.66.

2.1 Experimental Environment

All experiments were conducted under a controlled hardware and software environment to ensure fair comparison and reproducibility of the results. The experimental setup is summarized as follows:

CPU: Intel Core i7-13700H, 2400Mhz

RAM: 16 GB.

Storage: SSD 1TB.

Operating System: Windows 11 Pro .

Database System: Microsoft SQL Server 2019.

.NET Runtime: .NET 8.0.

Benchmarking Tool: BenchmarkDotNet version 0.14.0.

2.2 Database Design

A database is designed in SQL Server Management Studio (SSMS) 2019 with four tables created, including all types of relationships (one-to-one, one-to-many, and many-to-many), to simulate queries and CRUD operations. The database entities can be (see Fig. 1).

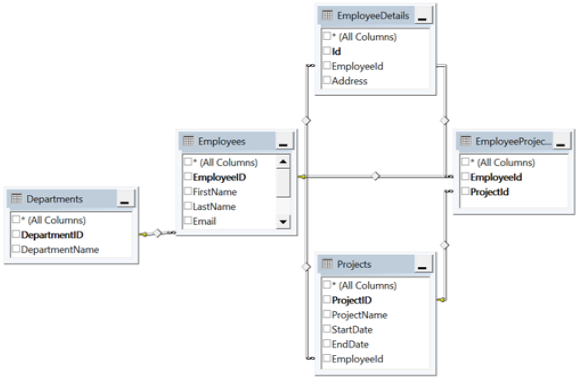

Fig. 1: Database Schema.

2.3 Application

ASP.NET Core web API application is created using .NET 8. The application is divided into two classes: one using Dapper and the other using EF Core.

EF Core provides a solution for an existing database through scaffold. Since EF Core uses models to access data, this application needs to create these models before it can start working with the database. The following commands are used

Scaffold-DbContext "Server = PRO-F-23; Database =EmployeeDB;Integrated Securi-ty=True; ConnectTimeout= 30;TrustServerCertificate=True"Microsoft.EntityFrameworkCore. SqlServer- OutputDir Models-Context Application Db Context -Force

A model is created from the database server and a context object to allow querying and saving the data, and this makes the application able to use EF Core. Dapper uses a different approach and only needs a connection string to be able to query the database. The following commands are used.

using var connection = new SqlConnection "Server=PRO-F-23; Database =EmployeeDB;IntegratedSecurity=True;ConnectTimeout=30;TrustServerCertificate=True ");

This contains the information that are needed to connect to the database. Inside this using statement, Dapper allows to save data and query the database using SQL statements. which are written manually.

2.4 Benchmarking Tool

Benchmark Dot Net is a tool for .NET that allows developers to benchmark performance in terms of execution time, memory, and CPU. It provides an option

to specify the number of times a query is run and the results are presented in readable files. Down-loaded by nuget.org.

2.5 Query

Complex queries will be used for testing, including add, update, delete, and fetch queries. The queries will be applied to one-to-one, one-to-many, and many-to-many relationships. Note that the queries will be applied to millions of records to demonstrate the significant differences between the two frameworks according to specific criteria.

3. RESULTS

This section presents the results gathered from the experiment. The results are divided into three different sections: execution time, memory consumption, and CPU usage when performing complex CRUD operations with data exceeding 1000000 records.

3.1 Execution Time

The execution times are shown when testing add, modify, delete, and fetch operations, and using all relationship types, for both Dapper and EF Core.

Figure 2 illustrates the execution time comparison between Dapper and EF Core for insert operations, showing that EF Core performs better (faster) in many-to-many operations, while Dapper is faster in one-to-one, one-to-many.

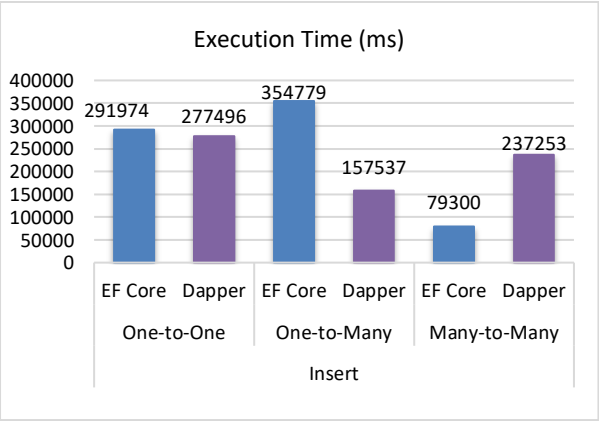

Fig. 2: Execution time for insert dapper vs EF Core.

Figure 3 illustrates the execution time comparison between Dapper and EF Core for update operations, showing that EF Core performs better (faster) in many-to-many operations, while Dapper is faster in one-to-one, one-to-many.

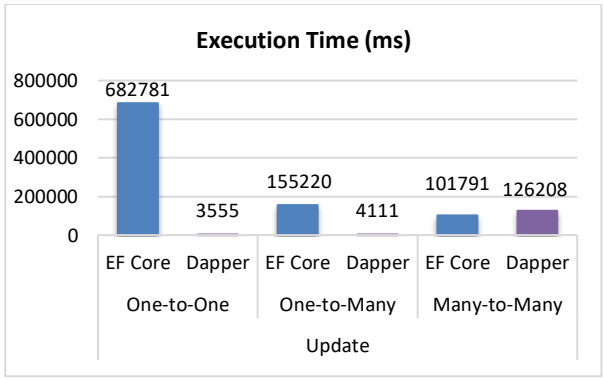

Fig. 3: Execution time for update dapper vs EF Core.

Figure 4 illustrates the execution time comparison between Dapper and EF Core for delete operations, showing that Dapper performs significantly faster than EF Core in one-to-one and many-to-many relationships, while EF Core shows better performance in one-to-many operations.

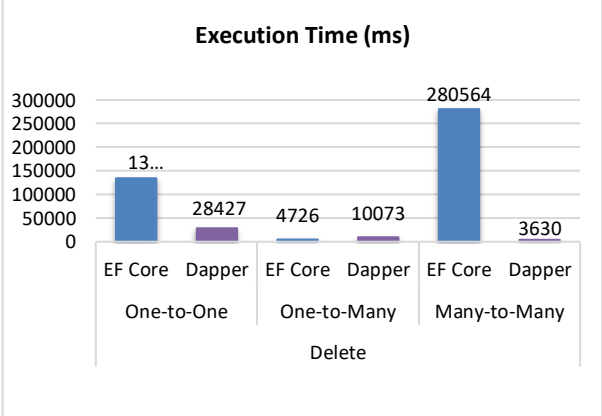

Fig. 4: Execution time for delete dapper vs EF Core.

Figure 5 illustrates the execution time comparison between Dapper and EF Core for GET operations, with Dapper proving faster across all relationship types.

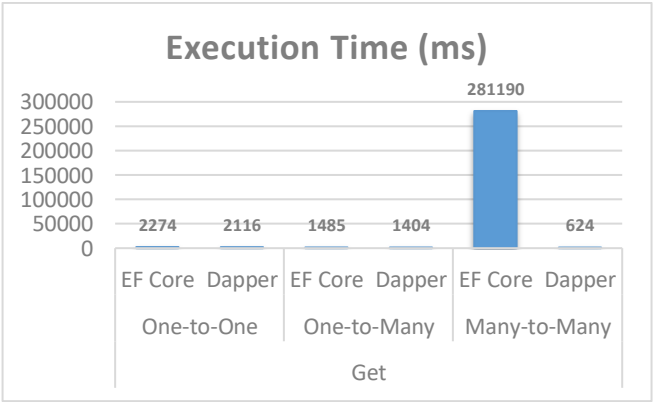

Fig. 5: Execution time for get dapper vs EF Core.

3.2 Memory Consumption

The Memory consumption metrics are shown when testing add, modify, delete, and fetch operations, and using all relationship types, for both Dapper and EF Core.

Figure 6 illustrates the memory consumption comparison between EF Core and Dapper, revealing that EF Core requires substantially more memory (up to gigabytes) for all operations, while Dapper maintains minimal memory usage (measured in kilobytes).

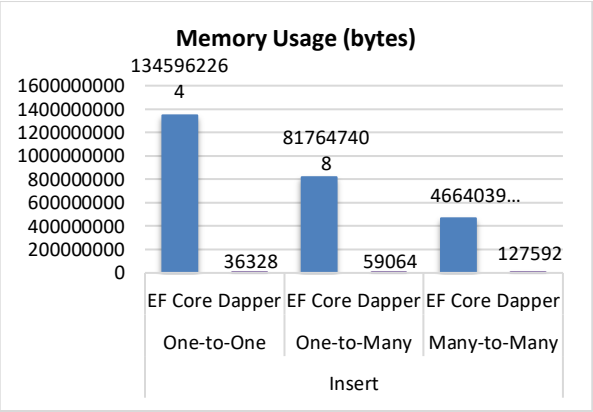

Fig. 6: Memory usage for insert dapper vs EF Core.

Figure 7 illustrates the memory usage comparison between EF Core and Dapper during update operations, showing EF Core consumes significantly more memory (reaching gigabytes) compared to Dapper's across all relationship.

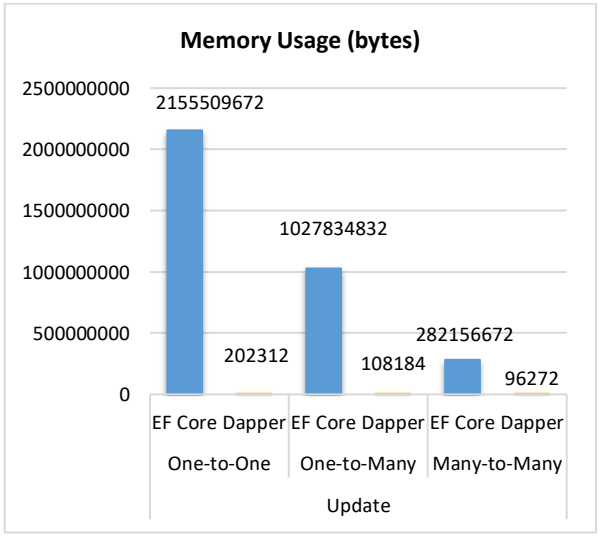

Fig. 7: Memory usage for update dapper vs EF Core.

Figure 8 illustrates the memory usage comparison between EF Core and Dapper during delete operations, showing EF Core consumes dramatically more memory compared to Dapper usage all relationship.

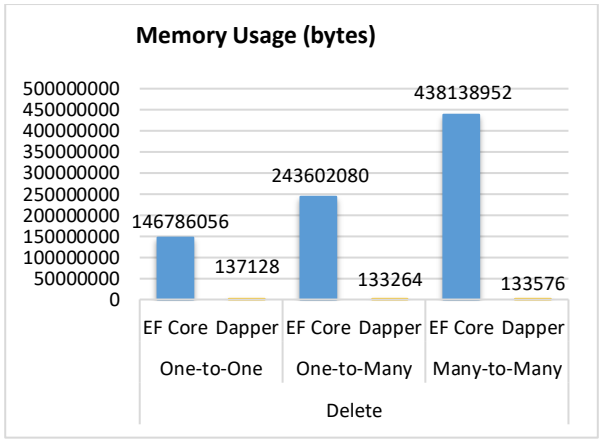

Fig. 8: Memory usage for delete dapper vs EF Core.

Figure 9 illustrates the memory usage comparison between EF Core and Dapper during GET operations, showing EF Core consumes significantly more memory (peaking at 1,481,672 bytes) compared to Dapper usage.

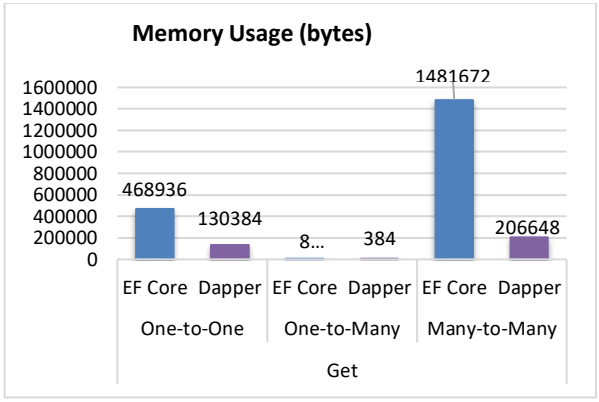

Fig. 9: Memory usage for get dapper vs EF Core.

3.3 CPU Usage

Shows the CPU usage when testing add, modify, delete, and fetch operations, using all relationship types, for both Dapper and EF Core.

Figure 10 illustrates CPU usage during insert operations, showing EF Core requires significantly more processing time (255,000-65000 ms) compared to Dapper.

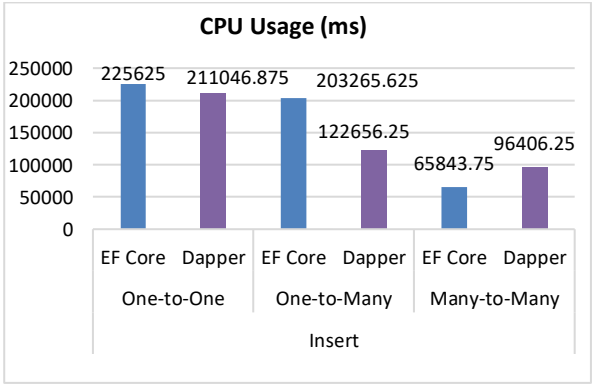

Fig. 10: CPU usage for insert dapper vs EF Core.

Figure 11 compares CPU usage for update operations, demonstrating Dapper's superior efficiency (as low as 12625ms) versus EF Core.

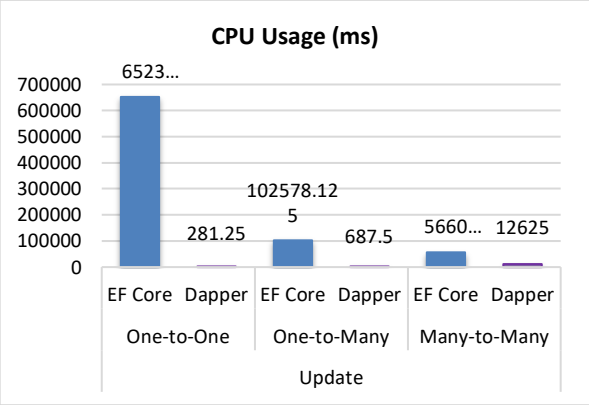

Fig. 11: CPU usage for update dapper vs EF Core.

Figure 12 illustrates CPU usage during delete operations, showing EF Core requires significantly more processing time (as low as 253390.625ms) compared to Dapper across all relationship types.

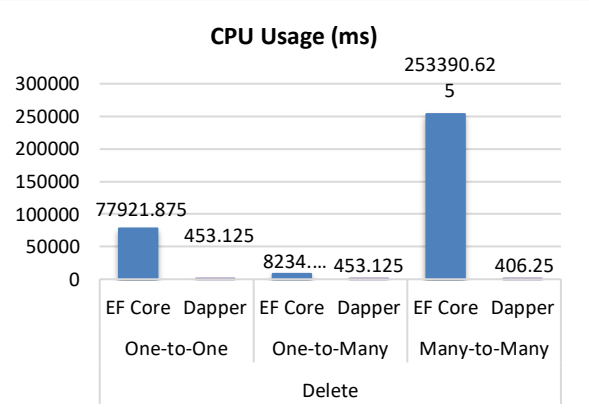

Fig. 12: CPU usage for delete dapper vs EF Core.

Figure 13 shows CPU usage for GET operations, where EF Core and Dapper similar perfor-mance in all relationship types.

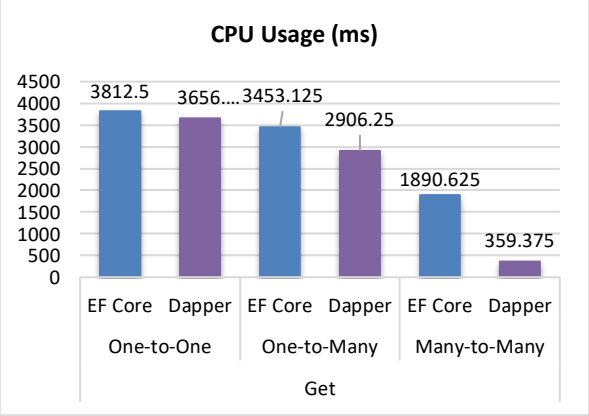

Fig. 13: CPU usage for get dapper vs EF Core.

4. DISCUSSION

As shown in (Figs. 1-5), Dapper achieved better execution times for most operations. Insertion, update, and select in 1:1 and 1:N relationships were faster using Dapper, while EF Core outperformed update and insertion in M:N relationships. Similar results can be found in,[1] which indicated that EF Core had faster execution times for both update and insertion when working with large inputs for this relation. Dapper outperformed EF Core for most relationships in the delete operation, similar to[1,2,8]. Then, the results of memory consumption were presented. As shown in (Figs. 6-9), EF Core allocated more memory across all operations and relationships compared to Dapper, which is an expected result. EF Core tracks and copies entities internally to track changes, which increases memory consumption. The size of the data being loaded also has a significant impact on memory consumption, similar to[1,6,8]. Finally, regarding CPU usage results. As shown in (Figs. 10-13), EF Core consumes a higher percentage of the processor compared to Dapper, especially when executing large queries or updating record sets at once. This is due to the fact that EF Core performs additional operations to complete a specific operation. As No Tracking can be used to reduce consumption and achieve bet-ter results. In contrast, Dapper operates more efficiently by executing queries directly without any additional processing, making it less CPU-intensive when data is very large.

5. Conclusion

Our performance analysis of Dapper and EF Core reveals a clear distinction in their efficiency when handling large-scale data operations. As a lightweight micro-ORM, Dapper's reliance on SQL queries enables superior speed in read, insert, update, and delete operations even across complex entity relationships. This makes it an optimal choice for high-performance applications processing millions of records, where execution speed is critical. Conversely EF Core excels in developer productivity, offering a more intuitive and flexible environment through LINQ and automatic relationship management. However, these conveniences incur trade-offs: EF Core exhibits higher memory and CPU usage, particularly during change tracking and automatically loading data. Ultimately, the decision between these tools hinges on project-specific priorities. Teams prioritizing performance and scalability may favor Dapper, while those valuing rapid development and maintainability might opt for EF Core.

REFERENCES

1. Wiatrowski T. Comparative analysis of ORM systems for the. NET platform. Journal of Computer Sciences Institute. 2024;31:97-102.
2. Yaba MS. Výkonnostní porovnání mezi Entity Framework Core 6 a Dapper.
3. Godinho A, ROSADO J, SA F, CARDOSO F. Performance comparison of RESTful web APIs using a test suite: .NET vs. Java Spring Boot. J Softw Syst Dev 2024:1-32.
4. Popa B, Popescu IM, Constantinescu RL, Albita A, Toma A, Cătălin-Vlădut M. Describing and Applying the Entity Framework from Simple Applications to Complex Scenarios. Case Study. In2024 25th International Carpathian Control Conference (ICCC) 2024 May 22 (pp. 01-06). IEEE.
5. Smith D. An exploration of the capability of a relational database management system to encompass business and persistence capabilities within architecturally layered software.
6. Myllyaho Forsberg M. An evaluation of. NET Object-Relational Mappers in relational databases: Entity Framework core and Dapper.
7. Zmaranda D, Pop-Fele LL, Gyorödi C, Gyorödi R, Pecherle G. Performance comparison of crud methods using net object relational mappers: A case study. International Journal of Advanced Computer Science and Applications. 2020;11.
8. Klimiuk M, Karabowicz P, Plechawska-Wójcik M. Comparative analysis of database mapping frameworks available in NuGet Manager. Journal of Computer Sciences Institute. 2024;32:231-8.
9. Güvercin AE, Avenoglu B. Performance analysis of object-relational mapping (orm) tools in. net 6 environment. Bilişim Teknolojileri Dergisi. 2022;15:453-65.
10. Jayaraman KD, Siddharth E. Harnessing the power of entity framework core for scalable database solutions. Journal of Quantum Science and Technology (JQST). 2025;2:151-71.
11. Milošević T, Lazarević SD. Merenje i analiza performansi Entity Core okvira i NHibernate okvira. InfoM – Časopis za informacione tehnologije i multimedijalne sisteme. 2022;(75–76):15–21.
12. Nilsson M. An evaluation of language integrated queries (LINQ) [Bachelor thesis]. 2022.